```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"
```

```c

#include "glist.h"
#include "gslice.h"
#include "gmessages.h"

#include "gtestutils.h"

/**
 * GList:
 * @data: holds the element's data, which can be a pointer to any kind
 *        of data, or any integer value using the
 *        [Type Conversion Macros][glib-Type-Conversion-Macros]
 * @next: contains the link to the next element in the list
 * @prev: contains the link to the previous element in the list
 *
 * The #GList struct is used for each element in a doubly-linked list.
 **/

/**
 * g_list_previous:
 * @list: an element in a #GList
 *
 * A convenience macro to get the previous element in a #GList.
 * Note that it is considered perfectly acceptable to access
 * @list->prev directly.
 *
 * Returns: the previous element, or %NULL if there are no previous
 *          elements
 **/

/**
 * g_list_next:
 * @list: an element in a #GList
 *
 * A convenience macro to get the next element in a #GList.
 * Note that it is considered perfectly acceptable to access
 * @list->next directly.
 *
 * Returns: the next element, or %NULL if there are no more elements
 **/

#define _g_list_alloc()         g_slice_new (GList)
#define _g_list_alloc0()        g_slice_new0 (GList)
#define _g_list_free1(list)     g_slice_free (GList, list)

/**
 * g_list_alloc:
 *
 * Allocates space for one #GList element. It is called by
 * g_list_append(), g_list_prepend(), g_list_insert() and
 * g_list_insert_sorted() and so is rarely used on its own.
 *
 * Returns: a pointer to the newly-allocated #GList element
 **/
GList *
g_list_alloc (void)
{
  return _g_list_alloc0 ();
}

/**
 * g_list_free:
 * @list: the first link of a #GList
 *
 * Frees all of the memory used by a #GList.
 * The freed elements are returned to the slice allocator.
 *
 * If list elements contain dynamically-allocated memory, you should
 * either use g_list_free_full() or free them manually first.
 *
 * It can be combined with g_steal_pointer() to ensure the list head pointer
 * is not left dangling:
 * |[<!-- language="C" -->
 * GList *list_of_borrowed_things = …;  /<!-- -->* (transfer container) *<!-- -->/
 * g_list_free (g_steal_pointer (&list_of_borrowed_things));
 * ]|
 */
void
g_list_free (GList *list)
{
  g_slice_free_chain (GList, list, next);
}

/**
 * g_list_free_1:
 * @list: a #GList element
 *
 * Frees one #GList element, but does not update links from the next and
 * previous elements in the list, so you should not call this function on an
 * element that is currently part of a list.
 *
 * It is usually used after g_list_remove_link().
 */
/**
 * g_list_free1:
 *
 * Another name for g_list_free_1().
 **/
void
g_list_free_1 (GList *list)
{
  _g_list_free1 (list);
}

/**
 * g_list_free_full:
 * @list: the first link of a #GList
 * @free_func: the function to be called to free each element's data
 *
 * Convenience method, which frees all the memory used by a #GList,
 * and calls @free_func on every element's data.
 *
 * @free_func must not modify the list (eg, by removing the freed
 * element from it).
 *
 * It can be combined with g_steal_pointer() to ensure the list head pointer
 * is not left dangling -- this also has the nice property that the head pointer
 * is cleared before any of the list elements are freed, to prevent double frees
 * from @free_func:
 * |[<!-- language="C" -->
 * GList *list_of_owned_things = …;  /<!-- -->* (transfer full) (element-type GObject) *<!-- -->/
 * g_list_free_full (g_steal_pointer (&list_of_owned_things), g_object_unref);
 * ]|
 *
 * Since: 2.28
 */
void
g_list_free_full (GList          *list,
                  GDestroyNotify  free_func)
{
  g_list_foreach (list, (GFunc) free_func, NULL);
  g_list_free (list);
}

/**
 * g_list_append:
 * @list: a pointer to a #GList
 * @data: the data for the new element
 *
 * Adds a new element on to the end of the list.
 *
 * Note that the return value is the new start of the list,
 * if @list was empty; make sure you store the new value.
 *
```

```c
 * g_list_append() has to traverse the entire list to find the end,
 * which is inefficient when adding multiple elements. A common idiom
 * to avoid the inefficiency is to use g_list_prepend() and reverse
 * the list with g_list_reverse() when all elements have been added.
 *
 * |[<!-- language="C" -->
 * // Notice that these are initialized to the empty list.
 * GList *string_list = NULL, *number_list = NULL;
 *
 * // This is a list of strings.
 * string_list = g_list_append (string_list, "first");
 * string_list = g_list_append (string_list, "second");
 *
 * // This is a list of integers.
 * number_list = g_list_append (number_list, GINT_TO_POINTER (27));
 * number_list = g_list_append (number_list, GINT_TO_POINTER (14));
 * ]|
 *
 * Returns: either @list or the new start of the #GList if @list was %NULL
 */
GList *
g_list_append (GList    *list,
               gpointer  data)
{
  GList *new_list;
  GList *last;

  new_list = _g_list_alloc ();
  new_list->data = data;
  new_list->next = NULL;

  if (list)
    {
      last = g_list_last (list);
      /* g_assert (last != NULL); */
      last->next = new_list;
      new_list->prev = last;

      return list;
    }
  else
    {
      new_list->prev = NULL;
      return new_list;
    }
}

/**
 * g_list_prepend:
 * @list: a pointer to a #GList, this must point to the top of the list
 * @data: the data for the new element
 *
 * Prepends a new element on to the start of the list.
 *
 * Note that the return value is the new start of the list,
 * which will have changed, so make sure you store the new value.
 *
 * |[<!-- language="C" -->
 * // Notice that it is initialized to the empty list.
 * GList *list = NULL;
 *
 * list = g_list_prepend (list, "last");
 * list = g_list_prepend (list, "first");
 * ]|
 *
 * Do not use this function to prepend a new element to a different
 * element than the start of the list. Use g_list_insert_before() instead.
 *
 * Returns: a pointer to the newly prepended element, which is the new
 *     start of the #GList
 */
GList *
g_list_prepend (GList    *list,
                gpointer  data)
```

```c
{
  GList *new_list;

  new_list = _g_list_alloc ();
  new_list->data = data;
  new_list->next = list;

  if (list)
    {
      new_list->prev = list->prev;
      if (list->prev)
        list->prev->next = new_list;
      list->prev = new_list;
    }
  else
    new_list->prev = NULL;

  return new_list;
}

/**
 * g_list_insert:
 * @list: a pointer to a #GList, this must point to the top of the list
 * @data: the data for the new element
 * @position: the position to insert the element. If this is
 *     negative, or is larger than the number of elements in the
 *     list, the new element is added on to the end of the list.
 *
 * Inserts a new element into the list at the given position.
 *
 * Returns: the (possibly changed) start of the #GList
 */
GList *
g_list_insert (GList    *list,
               gpointer  data,
               gint      position)
{
  GList *new_list;
  GList *tmp_list;

  if (position < 0)
    return g_list_append (list, data);
  else if (position == 0)
    return g_list_prepend (list, data);

  tmp_list = g_list_nth (list, position);
  if (!tmp_list)
    return g_list_append (list, data);

  new_list = _g_list_alloc ();
  new_list->data = data;
  new_list->prev = tmp_list->prev;
  tmp_list->prev->next = new_list;
  new_list->next = tmp_list;
  tmp_list->prev = new_list;

  return list;
}

/**
 * g_list_insert_before_link:
 * @list: a pointer to a #GList, this must point to the top of the list
 * @sibling: (nullable): the list element before which the new element
 *     is inserted or %NULL to insert at the end of the list
 * @link_: the list element to be added, which must not be part of
 *     any other list
 *
 * Inserts @link_ into the list before the given position.
 *
 * Returns: the (possibly changed) start of the #GList
 *
 * Since: 2.62
 */
GList *
g_list_insert_before_link (GList *list,
                           GList *sibling,
                           GList *link_)
{
  g_return_val_if_fail (link_ != NULL, list);
  g_return_val_if_fail (link_->prev == NULL, list);
  g_return_val_if_fail (link_->next == NULL, list);

  if (list == NULL)
    {
      g_return_val_if_fail (sibling == NULL, list);
      return link_;
    }
  else if (sibling != NULL)
    {
      link_->prev = sibling->prev;
      link_->next = sibling;
      sibling->prev = link_;
      if (link_->prev != NULL)
        {
          link_->prev->next = link_;
          return list;
        }
      else
        {
          g_return_val_if_fail (sibling == list, link_);
          return link_;
        }
    }
  else
    {
      GList *last;

      for (last = list; last->next != NULL; last = last->next) {}

      last->next = link_;
      last->next->prev = last;
      last->next->next = NULL;

      return list;
    }
}

/**
 * g_list_insert_before:
 * @list: a pointer to a #GList, this must point to the top of the list
 * @sibling: the list element before which the new element
 *     is inserted or %NULL to insert at the end of the list
 * @data: the data for the new element
 *
 * Inserts a new element into the list before the given position.
 *
 * Returns: the (possibly changed) start of the #GList
 */
GList *
g_list_insert_before (GList    *list,
                      GList    *sibling,
                      gpointer  data)
{
  if (list == NULL)
    {
      list = g_list_alloc ();
      list->data = data;
      g_return_val_if_fail (sibling == NULL, list);
      return list;
    }
  else if (sibling != NULL)
    {
      GList *node;

      node = _g_list_alloc ();
      node->data = data;
      node->prev = sibling->prev;
      node->next = sibling;
      sibling->prev = node;
      if (node->prev != NULL)
        {
          node->prev->next = node;
          return list;
        }
      else
        {
          g_return_val_if_fail (sibling == list, node);
          return node;
        }
    }
  else
```

```
    {
      GList *last;

      for (last = list; last->next != NULL; last = last->next) {}

      last->next = _g_list_alloc ();
      last->next->data = data;
      last->next->prev = last;
      last->next->next = NULL;

      return list;
    }
}

/**
 * g_list_concat:
 * @list1: a #GList, this must point to the top of the list
 * @list2: the #GList to add to the end of the first #GList,
 *     this must point  to the top of the list
 *
 * Adds the second #GList onto the end of the first #GList.
 * Note that the elements of the second #GList are not copied.
 * They are used directly.
 *
 * This function is for example used to move an element in the list.
 * The following example moves an element to the top of the list:
 * |[<!-- language="C" -->
 * list = g_list_remove_link (list, llink);
 * list = g_list_concat (llink, list);
 * ]|
 *
 * Returns: the start of the new #GList, which equals @list1 if not %NULL
 */
GList *
g_list_concat (GList *list1,
               GList *list2)
```

```c
{
  GList *tmp_list;

  if (list2)
    {
      tmp_list = g_list_last (list1);
      if (tmp_list)
        tmp_list->next = list2;
      else
        list1 = list2;
      list2->prev = tmp_list;
    }

  return list1;
}

static inline GList *
_g_list_remove_link (GList *list,
                     GList *link)
{
  if (link == NULL)
    return list;

  if (link->prev)
    {
      if (link->prev->next == link)
        link->prev->next = link->next;
      else
        g_warning ("corrupted double-linked list detected");
    }
  if (link->next)
    {
      if (link->next->prev == link)
        link->next->prev = link->prev;
      else
        g_warning ("corrupted double-linked list detected");
    }

  if (link == list)
    list = list->next;

  link->next = NULL;
  link->prev = NULL;

  return list;
}

/**
 * g_list_remove:
 * @list: a #GList, this must point to the top of the list
 * @data: the data of the element to remove
 *
 * Removes an element from a #GList.
 * If two elements contain the same data, only the first is removed.
 * If none of the elements contain the data, the #GList is unchanged.
 *
 * Returns: the (possibly changed) start of the #GList
 */
GList *
g_list_remove (GList         *list,
               gconstpointer  data)
{
  GList *tmp;

  tmp = list;
  while (tmp)
    {
      if (tmp->data != data)
        tmp = tmp->next;
      else
        {
          list = _g_list_remove_link (list, tmp);
          _g_list_free1 (tmp);

          break;
        }
    }
  return list;
}

/**
 * g_list_remove_all:
 * @list: a #GList, this must point to the top of the
```

```
list
 * @data: data to remove
 *
 * Removes all list nodes with data equal to @data.
 * Returns the new head of the list. Contrast with
 * g_list_remove() which removes only the first node
 * matching the given data.
 *
 * Returns: the (possibly changed) start of the #GList
 */
GList *
g_list_remove_all (GList         *list,
                   gconstpointer  data)
{
  GList *tmp = list;

  while (tmp)
    {
      if (tmp->data != data)
        tmp = tmp->next;
      else
        {
          GList *next = tmp->next;

          if (tmp->prev)
            tmp->prev->next = next;
          else
            list = next;
          if (next)
            next->prev = tmp->prev;

          _g_list_free1 (tmp);
          tmp = next;
        }
    }
  return list;
}

/**
 * g_list_remove_link:
 * @list: a #GList, this must point to the top of the list
 * @llink: an element in the #GList
 *
```

```c
 * Removes an element from a #GList, without freeing the element.
 * The removed element's prev and next links are set to %NULL, so
 * that it becomes a self-contained list with one element.
 *
 * This function is for example used to move an element in the list
 * (see the example for g_list_concat()) or to remove an element in
 * the list before freeing its data:
 * |[<!-- language="C" -->
 * list = g_list_remove_link (list, llink);
 * free_some_data_that_may_access_the_list_again (llink->data);
 * g_list_free (llink);
 * ]|
 *
 * Returns: the (possibly changed) start of the #GList
 */
GList *
g_list_remove_link (GList *list,
                    GList *llink)
{
  return _g_list_remove_link (list, llink);
}

/**
 * g_list_delete_link:
 * @list: a #GList, this must point to the top of the list
 * @link_: node to delete from @list
 *
 * Removes the node link_ from the list and frees it.
 * Compare this to g_list_remove_link() which removes the node
 * without freeing it.
 *
 * Returns: the (possibly changed) start of the #GList
 */
GList *
g_list_delete_link (GList *list,
                    GList *link_)
```

```c
{
  list = _g_list_remove_link (list, link_);
  _g_list_free1 (link_);

  return list;
}

/**
 * g_list_copy:
 * @list: a #GList, this must point to the top of the list
 *
 * Copies a #GList.
 *
 * Note that this is a "shallow" copy. If the list elements
 * consist of pointers to data, the pointers are copied but
 * the actual data is not. See g_list_copy_deep() if you need
 * to copy the data as well.
 *
 * Returns: the start of the new list that holds the same data as @list
 */
GList *
g_list_copy (GList *list)
{
  return g_list_copy_deep (list, NULL, NULL);
}

/**
 * g_list_copy_deep:
 * @list: a #GList, this must point to the top of the list
 * @func: (scope call): a copy function used to copy every element in the list
 * @user_data: user data passed to the copy function @func, or %NULL
 *
 * Makes a full (deep) copy of a #GList.
 *
 * In contrast with g_list_copy(), this function uses @func to make
 * a copy of each list element, in addition to copying the list
 * container itself.
 *
 * @func, as a #GCopyFunc, takes two arguments, the data to be copied
 * and a @user_data pointer. On common processor architectures, it's safe to
 * pass %NULL as @user_data if the copy function takes only one argument. You
 * may get compiler warnings from this though if compiling with GCC’s
 * `-Wcast-function-type` warning.
 *
 * For instance, if @list holds a list of GObjects, you can do:
 * |[<!-- language="C" -->
 * another_list = g_list_copy_deep (list, (GCopyFunc) g_object_ref, NULL);
 * ]|
 *
 * And, to entirely free the new list, you could do:
 * |[<!-- language="C" -->
 * g_list_free_full (another_list, g_object_unref);
 * ]|
 *
 * Returns: the start of the new list that holds a full copy of @list,
 *     use g_list_free_full() to free it
 *
 * Since: 2.34
 */
GList *
g_list_copy_deep (GList     *list,
                  GCopyFunc  func,
                  gpointer   user_data)
{
  GList *new_list = NULL;

  if (list)
    {
      GList *last;

      new_list = _g_list_alloc ();
      if (func)
        new_list->data = func (list->data, user_data);
      else
        new_list->data = list->data;
      new_list->prev = NULL;
      last = new_list;
      list = list->next;
      while (list)
        {
          last->next = _g_list_alloc ();
          last->next->prev = last;
          last = last->next;
          if (func)
            last->data = func (list->data, user_data);
          else
            last->data = list->data;
          list = list->next;
        }
      last->next = NULL;
    }

  return new_list;
}

/**
 * g_list_reverse:
 * @list: a #GList, this must point to the top of the list
 *
 * Reverses a #GList.
 * It simply switches the next and prev pointers of each element.
 *
 * Returns: the start of the reversed #GList
 */
GList *
g_list_reverse (GList *list)
{
  GList *last;

  last = NULL;
  while (list)
    {
      last = list;
      list = last->next;
      last->next = last->prev;
      last->prev = list;
    }

  return last;
}

/**
 * g_list_nth:
 * @list: a #GList, this must point to the top of the list
 * @n: the position of the element, counting from 0
 *
 * Gets the element at the given position in a #GList.
 *
 * This iterates over the list until it reaches the @n-th position. If you
 * intend to iterate over every element, it is better to use a for-loop as
 * described in the #GList introduction.
 *
 * Returns: the element, or %NULL if the position is off
 *     the end of the #GList
 */
GList *
g_list_nth (GList *list,
            guint  n)
{
  while ((n-- > 0) && list)
    list = list->next;

  return list;
}

/**
 * g_list_nth_prev:
 * @list: a #GList
 * @n: the position of the element, counting from 0
 *
 * Gets the element @n places before @list.
 *
 * Returns: the element, or %NULL if the position is
 *     off the end of the #GList
 */
GList *
g_list_nth_prev (GList *list,
                 guint  n)
{
  while ((n-- > 0) && list)
    list = list->prev;

  return list;
}

/**
 * g_list_nth_data:
 * @list: a #GList, this must point to the top of the list
 * @n: the position of the element
 *
 * Gets the data of the element at the given position.
 *
 * This iterates over the list until it reaches the @n-th position. If you
 * intend to iterate over every element, it is better to use a for-loop as
 * described in the #GList introduction.
 *
 * Returns: the element's data, or %NULL if the position
 *     is off the end of the #GList
 */
gpointer
g_list_nth_data (GList *list,
                 guint  n)
{
  while ((n-- > 0) && list)
    list = list->next;

  return list ? list->data : NULL;
}

/**
 * g_list_find:
 * @list: a #GList, this must point to the top of the list
 * @data: the element data to find
 *
```

```c
 * Finds the element in a #GList which contains the given data.
 *
 * Returns: the found #GList element, or %NULL if it is not found
 */
GList *
g_list_find (GList         *list,
             gconstpointer  data)
{
  while (list)
    {
      if (list->data == data)
        break;
      list = list->next;
    }

  return list;
}

/**
 * g_list_find_custom:
 * @list: a #GList, this must point to the top of the list
 * @data: user data passed to the function
 * @func: (scope call): the function to call for each element.
 *     It should return 0 when the desired element is found
 *
 * Finds an element in a #GList, using a supplied function to
 * find the desired element. It iterates over the list, calling
 * the given function which should return 0 when the desired
 * element is found. The function takes two #gconstpointer arguments,
 * the #GList element's data as the first argument and the
 * given user data.
 *
 * Returns: the found #GList element, or %NULL if it is not found
 */
GList *
g_list_find_custom (GList         *list,
                    gconstpointer  data,
                    GCompareFunc   func)
{
  g_return_val_if_fail (func != NULL, list);

  while (list)
    {
      if (! func (list->data, data))
        return list;
      list = list->next;
    }

  return NULL;
}

/**
 * g_list_position:
 * @list: a #GList, this must point to the top of the list
 * @llink: an element in the #GList
 *
 * Gets the position of the given element
 * in the #GList (starting from 0).
 *
 * Returns: the position of the element in the #GList,
 *     or -1 if the element is not found
 */
gint
g_list_position (GList *list,
                 GList *llink)
{
  gint i;

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

  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
 */
```

```c
GList *
g_list_last (GList *list)
{
  if (list)
    {
      while (list->next)
        list = list->next;
    }

  return list;
}

/**
 * g_list_first:
 * @list: any #GList element
 *
 * Gets the first element in a #GList.
 *
 * Returns: the first element in the #GList,
 *     or %NULL if the #GList has no elements
 */
GList *
g_list_first (GList *list)
{
  if (list)
    {
      while (list->prev)
        list = list->prev;
    }

  return list;
}

/**
 * g_list_length:
 * @list: a #GList, this must point to the top of the list
 *
 * Gets the number of elements in a #GList.
 *
 * This function iterates over the whole list to count its elements.
 * Use a #GQueue instead of a GList if you regularly need the number
  * of items. To check whether the list is non-empty, it is faster to check
  * @list against %NULL.
  *
  * Returns: the number of elements in the #GList
  */
 guint
 g_list_length (GList *list)
 {
   guint length;

   length = 0;
   while (list)
     {
       length++;
       list = list->next;
     }

   return length;
 }

 /**
  * g_list_foreach:
  * @list: a #GList, this must point to the top of the list
  * @func: (scope call): the function to call with each element's data
  * @user_data: user data to pass to the function
  *
  * Calls a function for each element of a #GList.
  *
  * It is safe for @func to remove the element from @list, but it must
  * not modify any part of the list after that element.
  */
 /**
  * GFunc:
  * @data: the element's data
  * @user_data: user data passed to g_list_foreach() or g_slist_foreach()
  *
  * Specifies the type of functions passed to g_list_foreach() and
  * g_slist_foreach().
  */
 void
 g_list_foreach (GList    *list,
                 GFunc     func,
                 gpointer  user_data)
 {
   while (list)
     {
       GList *next = list->next;
       (*func) (list->data, user_data);
       list = next;
     }
 }

 static GList*
 g_list_insert_sorted_real (GList    *list,
                            gpointer  data,
                            GFunc     func,
                            gpointer  user_data)
 {
   GList *tmp_list = list;
   GList *new_list;
   gint cmp;

   g_return_val_if_fail (func != NULL, list);

   if (!list)
     {
       new_list = _g_list_alloc0 ();
       new_list->data = data;
       return new_list;
     }

   cmp = ((GCompareDataFunc) func) (data, tmp_list->data, user_data);

   while ((tmp_list->next) && (cmp > 0))
     {
       tmp_list = tmp_list->next;

       cmp = ((GCompareDataFunc) func) (data, tmp_list->data, user_data);
     }

```

```c
  new_list = _g_list_alloc0 ();
  new_list->data = data;

  if ((!tmp_list->next) && (cmp > 0))
    {
      tmp_list->next = new_list;
      new_list->prev = tmp_list;
      return list;
    }

  if (tmp_list->prev)
    {
      tmp_list->prev->next = new_list;
      new_list->prev = tmp_list->prev;
    }
  new_list->next = tmp_list;
  tmp_list->prev = new_list;

  if (tmp_list == list)
    return new_list;
  else
    return list;
}

/**
 * g_list_insert_sorted:
 * @list: a pointer to a #GList, this must point to the top of the
 *     already sorted list
 * @data: the data for the new element
 * @func: (scope call): the function to compare elements in the list. It should
 *     return a number > 0 if the first parameter comes after the
 *     second parameter in the sort order.
 *
 * Inserts a new element into the list, using the given comparison
 * function to determine its position.
 *
 * If you are adding many new elements to a list, and the number of
 * new elements is much larger than the length of the list, use
 * g_list_prepend() to add the new items and sort the list afterwards
 * with g_list_sort().
 *
 * Returns: the (possibly changed) start of the #GList
 */
GList *
g_list_insert_sorted (GList        *list,
                      gpointer      data,
                      GCompareFunc  func)
{
  return g_list_insert_sorted_real (list, data, (GFunc) func, NULL);
}

/**
 * g_list_insert_sorted_with_data:
 * @list: a pointer to a #GList, this must point to the top of the
 *     already sorted list
 * @data: the data for the new element
 * @func: (scope call): the function to compare elements in the list. It should
 *     return a number > 0 if the first parameter  comes after the
 *     second parameter in the sort order.
 * @user_data: user data to pass to comparison function
 *
 * Inserts a new element into the list, using the given comparison
 * function to determine its position.
 *
 * If you are adding many new elements to a list, and the number of
 * new elements is much larger than the length of the list, use
 * g_list_prepend() to add the new items and sort the list afterwards
 * with g_list_sort().
 *
 * Returns: the (possibly changed) start of the #GList
 *
 * Since: 2.10
 */
```

```c
GList *
g_list_insert_sorted_with_data (GList            *list,
                                gpointer          data,
                                GCompareDataFunc  func,
                                gpointer          user_data)
{
  return g_list_insert_sorted_real (list, data, (GFunc) func, user_data);
}

static GList *
g_list_sort_merge (GList     *l1,
                   GList     *l2,
                   GFunc     compare_func,
                   gpointer  user_data)
{
  GList list, *l, *lprev;
  gint cmp;

  l = &list;
  lprev = NULL;

  while (l1 && l2)
    {
      cmp = ((GCompareDataFunc) compare_func) (l1->data, l2->data, user_data);

      if (cmp <= 0)
        {
          l->next = l1;
          l1 = l1->next;
        }
      else
        {
          l->next = l2;
          l2 = l2->next;
        }
      l = l->next;
      l->prev = lprev;
      lprev = l;
    }
  l->next = l1 ? l1 : l2;
  l->next->prev = l;

  return list.next;
}

static GList *
g_list_sort_real (GList    *list,
                  GFunc     compare_func,
                  gpointer  user_data)
{
  GList *l1, *l2;

  if (!list)
    return NULL;
  if (!list->next)
    return list;

  l1 = list;
  l2 = list->next;

  while ((l2 = l2->next) != NULL)
    {
      if ((l2 = l2->next) == NULL)
        break;
      l1 = l1->next;
    }
  l2 = l1->next;
  l1->next = NULL;

  return g_list_sort_merge (g_list_sort_real (list, compare_func, user_data),
                            g_list_sort_real (l2, compare_func, user_data),
                            compare_func,
                            user_data);
}

/**
 * g_list_sort:
 * @list: a #GList, this must point to the top of the list
 * @compare_func: (scope call): the comparison function used to sort the #GList.
 *     This function is passed the data from 2 elements of the #GList
 *     and should return 0 if they are equal, a negative value if the
 *     first element comes before the second, or a positive value if
 *     the first element comes after the second.
 *
 * Sorts a #GList using the given comparison function. The algorithm
 * used is a stable sort.
 *
 * Returns: the (possibly changed) start of the #GList
 */
/**
 * GCompareFunc:
 * @a: a value
 * @b: a value to compare with
 *
 * Specifies the type of a comparison function used to compare two
 * values.  The function should return a negative integer if the first
 * value comes before the second, 0 if they are equal, or a positive
 * integer if the first value comes after the second.
 *
 * Returns: negative value if @a < @b; zero if @a = @b; positive
 *          value if @a > @b
 */
GList *
g_list_sort (GList        *list,
             GCompareFunc  compare_func)
{
  return g_list_sort_real (list, (GFunc) compare_func, NULL);
}

/**
 * g_list_sort_with_data:
 * @list: a #GList, this must point to the top of the list
 * @compare_func: (scope call): comparison function
 * @user_data: user data to pass to comparison function
 *
 * Like g_list_sort(), but the comparison function accepts
 * a user data argument.
 *
 * Returns: the (possibly changed) start of the #GList
 */
/**
 * GCompareDataFunc:
 * @a: a value
 * @b: a value to compare with
 * @user_data: user data
 *
 * Specifies the type of a comparison function used to compare two
 * values.  The function should return a negative integer if the first
 * value comes before the second, 0 if they are equal, or a positive
 * integer if the first value comes after the second.
 *
 * Returns: negative value if @a < @b; zero if @a = @b; positive
 *          value if @a > @b
 */
GList *
g_list_sort_with_data (GList            *list,
                       GCompareDataFunc  compare_func,
                       gpointer          user_data)
{
  return g_list_sort_real (list, (GFunc) compare_func, user_data);
}

/**
 * g_clear_list: (skip)
 * @list_ptr: (not nullable): a #GList return location
 * @destroy: (nullable): the function to pass to g_list_free_full() or %NULL to not free elements
 *
 * Clears a pointer to a #GList, freeing it and, optionally, freeing its elements using @destroy.
 *
 * @list_ptr must be a valid pointer. If @list_ptr points to a null #GList, this does nothing.
 *
 * Since: 2.64
 */
void
(g_clear_list) (GList          **list_ptr,
                GDestroyNotify   destroy)
{
  GList *list;

  list = *list_ptr;
  if (list)
    {
      *list_ptr = NULL;

      if (destroy)
        g_list_free_full (list, destroy);
      else
        g_list_free (list);
    }
}

```

```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)
```

```
#error "Only <glib.h> can be included directly."
#endif

#include <glib/gmem.h>
#include <glib/gnode.h>

G_BEGIN_DECLS

typedef struct _GList GList;

struct _GList
{
  gpointer data;
  GList *next;
  GList *prev;
};

/* Doubly linked lists
 */
GLIB_AVAILABLE_IN_ALL
GList*   g_list_alloc                   (void)
G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
void     g_list_free                    (GList
            *list);
GLIB_AVAILABLE_IN_ALL
void     g_list_free_1                  (GList
            *list);
#define  g_list_free1                   g_list_free_1
GLIB_AVAILABLE_IN_ALL
void     g_list_free_full               (GList
            *list,
                     GDestroyNotify    free_func);
GLIB_AVAILABLE_IN_ALL
GList*   g_list_append                  (GList
            *list,
                     gpointer          data)
G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
GList*   g_list_prepend                 (GList
            *list,
                     gpointer          data)
G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
```

```
GList*   g_list_insert                  (GList
            *list,
                     gpointer          data,
                     gint              position)
G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
GList*   g_list_insert_sorted           (GList
            *list,
                     gpointer          data,
                     GCompareFunc      func)
G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
GList*   g_list_insert_sorted_with_data (GList
            *list,
                     gpointer          data,
                     GCompareDataFunc  func,
                     gpointer          user_data)
G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
GList*   g_list_insert_before           (GList
            *list,
                     GList            *sibling,
                     gpointer          data)
G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_2_62
GList*   g_list_insert_before_link      (GList
            *list,
                     GList            *sibling,
                     GList            *link_)
G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
GList*   g_list_concat                  (GList
            *list1,
                     GList            *list2)
G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
GList*   g_list_remove                  (GList
            *list,
                     gconstpointer     data)
G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
GList*   g_list_remove_all              (GList
            *list,
                     gconstpointer     data)
```

```
G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
GList*   g_list_remove_link             (GList *list,
                     GList            *llink) G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
GList*   g_list_delete_link             (GList *list,
                     GList            *link_) G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
GList*   g_list_reverse                 (GList *list) G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
GList*   g_list_copy                    (GList *list) G_GNUC_WARN_UNUSED_RESULT;

GLIB_AVAILABLE_IN_2_34
GList*   g_list_copy_deep               (GList *list,
                     GCopyFunc         func,
                     gpointer          user_data) G_GNUC_WARN_UNUSED_RESULT;

GLIB_AVAILABLE_IN_ALL
GList*   g_list_nth                     (GList *list,
                     guint             n);
GLIB_AVAILABLE_IN_ALL
GList*   g_list_nth_prev                (GList *list,
                     guint             n);
GLIB_AVAILABLE_IN_ALL
GList*   g_list_find                    (GList *list,
                     gconstpointer     data);
GLIB_AVAILABLE_IN_ALL
GList*   g_list_find_custom             (GList *list,
                     gconstpointer     data,
                     GCompareFunc      func);
GLIB_AVAILABLE_IN_ALL
gint     g_list_position                (GList
            *list,
                     GList            *llink);
GLIB_AVAILABLE_IN_ALL
gint     g_list_index                   (GList
            *list,
                     gconstpointer     data);
GLIB_AVAILABLE_IN_ALL
GList*   g_list_last                    (GList
            *list);
GLIB_AVAILABLE_IN_ALL
GList*   g_list_first                   (GList
            *list);
GLIB_AVAILABLE_IN_ALL
guint    g_list_length                  (GList
            *list);
GLIB_AVAILABLE_IN_ALL
void     g_list_foreach                 (GList
            *list,
                     GFunc             func,
                     gpointer          user_data);
GLIB_AVAILABLE_IN_ALL
GList*   g_list_sort                    (GList
            *list,
                     GCompareFunc      compare_func)
G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
GList*   g_list_sort_with_data          (GList
            *list,
                     GCompareDataFunc  compare_func,
                     gpointer          user_data)
G_GNUC_WARN_UNUSED_RESULT;
GLIB_AVAILABLE_IN_ALL
gpointer g_list_nth_data                (GList
            *list,
                     guint             n);

GLIB_AVAILABLE_IN_2_64
void     g_clear_list                   (GList
           **list_ptr,
                                         GDestroyNotify
    destroy);

#define  g_clear_list(list_ptr, destroy)       \
  G_STMT_START {                               \
```

```c
    GList *_list;                              \
                                               \
    _list = *(list_ptr);                       \
    if (_list)                                 \
      {                                        \
        *list_ptr = NULL;                      \
                                               \
        if ((destroy) != NULL)                 \
          g_list_free_full (_list, (destroy)); \
        else                                   \
          g_list_free (_list);                 \
      }                                        \
  } G_STMT_END                                 \
  GLIB_AVAILABLE_MACRO_IN_2_64


#define g_list_previous(list)           ((list) ? (((GList *)(list))->prev) : NULL)
#define g_list_next(list)               ((list) ? (((GList *)(list))->next) : NULL)

G_END_DECLS

#endif /* __G_LIST_H__ */

```

```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;
}
static inline bool __list_del_entry_valid(struct list_head *entry)
{
    return true;
}
#endif

/*
 * Insert a new entry between two known consecutive entries.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static inline void __list_add(struct list_head *new,
                  struct list_head *prev,
                  struct list_head *next)
{
    if (!__list_add_valid(new, prev, next))
        return;

    next->prev = new;
    new->next = next;
    new->prev = prev;
    WRITE_ONCE(prev->next, new);
}

/**
 * list_add - add a new entry
 * @new: new entry to be added
 * @head: list head to add it after
 *
 * Insert a new entry after the specified head.
 * This is good for implementing stacks.
 */
static inline void list_add(struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}


```

```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);
}
```

```c

static inline void list_del(struct list_head *entry)
{
    __list_del_entry(entry);
    entry->next = LIST_POISON1;
    entry->prev = LIST_POISON2;
}

/**
 * list_replace - replace old entry by new one
 * @old : the element to be replaced
 * @new : the new element to insert
 *
 * If @old was empty, it will be overwritten.
 */
static inline void list_replace(struct list_head *old,
                struct list_head *new)
{
    new->next = old->next;
    new->next->prev = new;
    new->prev = old->prev;
    new->prev->next = new;
}

static inline void list_replace_init(struct list_head *old,
                    struct list_head *new)
{
    list_replace(old, new);
    INIT_LIST_HEAD(old);
}

/**
 * list_del_init - deletes entry from list and reinitialize it.
 * @entry: the element to delete from the list.
 */
static inline void list_del_init(struct list_head *entry)
{
    __list_del_entry(entry);
    INIT_LIST_HEAD(entry);
}

/**
```

```
 * list_move - delete from one list and add as another's head
 * @list: the entry to move
 * @head: the head that will precede our entry
 */
static inline void list_move(struct list_head *list, struct list_head *head)
{
    __list_del_entry(list);
    list_add(list, head);
}

/**
 * list_move_tail - delete from one list and add as another's tail
 * @list: the entry to move
 * @head: the head that will follow our entry
 */
static inline void list_move_tail(struct list_head *list,
                  struct list_head *head)
{
    __list_del_entry(list);
    list_add_tail(list, head);
}

/**
 * list_is_last - tests whether @list is the last entry in list @head
 * @list: the entry to test
 * @head: the head of the list
 */
static inline int list_is_last(const struct list_head *list,
                const struct list_head *head)
{
    return list->next == head;
}

/**
 * list_empty - tests whether a list is empty
 * @head: the list to test.
 */
static inline int list_empty(const struct list_head *head)
```

```c
{
    return READ_ONCE(head->next) == head;
}

/**
 * list_empty_careful - tests whether a list is empty and not being modified
 * @head: the list to test
 *
 * Description:
 * tests whether a list is empty _and_ checks that no other CPU might be
 * in the process of modifying either member (next or prev)
 *
 * NOTE: using list_empty_careful() without synchronization
 * can only be safe if the only activity that can happen
 * to the list entry is list_del_init(). Eg. it cannot be used
 * if another CPU could re-list_add() it.
 */
static inline int list_empty_careful(const struct list_head *head)
{
    struct list_head *next = head->next;
    return (next == head) && (next == head->prev);
}

/**
 * list_rotate_left - rotate the list to the left
 * @head: the head of the list
 */
static inline void list_rotate_left(struct list_head *head)
{
    struct list_head *first;

    if (!list_empty(head)) {
        first = head->next;
        list_move_tail(first, head);
    }
}

```

```c
/**
 * list_is_singular - tests whether a list has just one entry.
 * @head: the list to test.
 */
static inline int list_is_singular(const struct list_head *head)
{
    return !list_empty(head) && (head->next == head->prev);
}

static inline void __list_cut_position(struct list_head *list,
        struct list_head *head, struct list_head *entry)
{
    struct list_head *new_first = entry->next;
    list->next = head->next;
    list->next->prev = list;
    list->prev = entry;
    entry->next = list;
    head->next = new_first;
    new_first->prev = head;
}

/**
 * list_cut_position - cut a list into two
 * @list: a new list to add all removed entries
 * @head: a list with entries
 * @entry: an entry within head, could be the head itself
 *  and if so we won't cut the list
 *
 * This helper moves the initial part of @head, up to and
 * including @entry, from @head to @list. You should
 * pass on @entry an element you know is on @head. @list
 * should be an empty list or a list you do not care about
 * losing its data.
 *
 */
static inline void list_cut_position(struct list_head *list,
        struct list_head *head, struct list_head *entry)
{
    if (list_empty(head))
        return;
    if (list_is_singular(head) &&
        (head->next != entry && head != entry))
        return;
    if (entry == head)
        INIT_LIST_HEAD(list);
    else
        __list_cut_position(list, head, entry);
}

static inline void __list_splice(const struct list_head *list,
                 struct list_head *prev,
                 struct list_head *next)
{
    struct list_head *first = list->next;
    struct list_head *last = list->prev;

    first->prev = prev;
    prev->next = first;

    last->next = next;
    next->prev = last;
}

/**
 * list_splice - join two lists, this is designed for stacks
 * @list: the new list to add.
 * @head: the place to add it in the first list.
 */
static inline void list_splice(const struct list_head *list,
                struct list_head *head)
{
    if (!list_empty(list))
        __list_splice(list, head, head->next);
}

/**
 * list_splice_tail - join two lists, each list being a queue
 * @list: the new list to add.
 * @head: the place to add it in the first list.
 */
static inline void list_splice_tail(struct list_head *list,
                struct list_head *head)
{
    if (!list_empty(list))
        __list_splice(list, head->prev, head);
}

/**
 * list_splice_init - join two lists and reinitialise the emptied list.
 * @list: the new list to add.
 * @head: the place to add it in the first list.
 *
 * The list at @list is reinitialised
 */
static inline void list_splice_init(struct list_head *list,
                    struct list_head *head)
{
    if (!list_empty(list)) {
        __list_splice(list, head, head->next);
        INIT_LIST_HEAD(list);
    }
}

/**
 * list_splice_tail_init - join two lists and reinitialise the emptied list
 * @list: the new list to add.
 * @head: the place to add it in the first list.
 *
 * Each of the lists is a queue.
 * The list at @list is reinitialised
 */
static inline void list_splice_tail_init(struct list_head *list,
                     struct list_head *head)
{
    if (!list_empty(list)) {
        __list_splice(list, head->prev, head);
        INIT_LIST_HEAD(list);
    }
}

/**
 * list_entry - get the struct for this entry
 * @ptr:    the &struct list_head pointer.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 */
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)

/**
 * list_first_entry - get the first element from a list
 * @ptr:    the list head to take the element from.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 *
 * Note, that list is expected to be not empty.
 */
#define list_first_entry(ptr, type, member) \
    list_entry((ptr)->next, type, member)

/**
 * list_last_entry - get the last element from a list
 * @ptr:    the list head to take the element from.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 *
 * Note, that list is expected to be not empty.
 */
#define list_last_entry(ptr, type, member) \
    list_entry((ptr)->prev, type, member)

/**
 * list_first_entry_or_null - get the first element from a list
 * @ptr:    the list head to take the element from.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 *
 * Note that if the list is empty, it returns NULL.
 */
#define list_first_entry_or_null(ptr, type, member) ({ \
    struct list_head *head__ = (ptr); \
    struct list_head *pos__ = READ_ONCE(head__->next); \
    pos__ != head__ ? list_entry(pos__, type, member) : NULL; \
})

/**
 * list_next_entry - get the next element in list
 * @pos:    the type * to cursor
 * @member: the name of the list_head within the struct.
 */
#define list_next_entry(pos, member) \
    list_entry((pos)->member.next, typeof(*(pos)), member)

/**
 * list_prev_entry - get the prev element in list
 * @pos:    the type * to cursor
 * @member: the name of the list_head within the struct.
 */
#define list_prev_entry(pos, member) \
    list_entry((pos)->member.prev, typeof(*(pos)), member)

/**
 * list_for_each-   iterate over a list
 * @pos:    the &struct list_head to use as a loop cursor.
 * @head:   the head for your list.
 */
#define list_for_each(pos, head) \
    for (pos = (head)->next; pos != (head); pos = pos->next)

/**
 * list_for_each_prev   -   iterate over a list backwards
 * @pos:    the &struct list_head to use as a loop cursor.
 * @head:   the head for your list.
 */
#define list_for_each_prev(pos, head) \
    for (pos = (head)->prev; pos != (head); pos = pos->prev)

/**
 * list_for_each_safe - iterate over a list safe against
```

```
removal of list entry
 * @pos:	the &struct list_head to use as a loop cursor.
 * @n:		another &struct list_head to use as temporary storage
 * @head:	the head for your list.
 */
#define list_for_each_safe(pos, n, head) \
	for (pos = (head)->next, n = pos->next; pos != (head); \
		pos = n, n = pos->next)

/**
 * list_for_each_prev_safe - iterate over a list backwards safe against removal of list entry
 * @pos:	the &struct list_head to use as a loop cursor.
 * @n:		another &struct list_head to use as temporary storage
 * @head:	the head for your list.
 */
#define list_for_each_prev_safe(pos, n, head) \
	for (pos = (head)->prev, n = pos->prev; \
	     pos != (head); \
	     pos = n, n = pos->prev)

/**
 * list_for_each_entry	-	iterate over list of given type
 * @pos:	the type * to use as a loop cursor.
 * @head:	the head for your list.
 * @member:	the name of the list_head within the struct.
 */
#define list_for_each_entry(pos, head, member)				\
	for (pos = list_first_entry(head, typeof(*pos), member);	\
	     &pos->member != (head);					\
	     pos = list_next_entry(pos, member))

/**
 * list_for_each_entry_reverse - iterate backwards over list of given type.
 * @pos:	the type * to use as a loop cursor.
 * @head:	the head for your list.
 * @member:	the name of the list_head within the struct.
 */
#define list_for_each_entry_reverse(pos, head, member)			\
	for (pos = list_last_entry(head, typeof(*pos), member);		\
	     &pos->member != (head); 					\
	     pos = list_prev_entry(pos, member))

/**
 * list_prepare_entry - prepare a pos entry for use in list_for_each_entry_continue()
 * @pos:	the type * to use as a start point
 * @head:	the head of the list
 * @member:	the name of the list_head within the struct.
 *
 * Prepares a pos entry for use as a start point in list_for_each_entry_continue().
 */
#define list_prepare_entry(pos, head, member) \
	((pos) ? : list_entry(head, typeof(*pos), member))

/**
 * list_for_each_entry_continue - continue iteration over list of given type
 * @pos:	the type * to use as a loop cursor.
 * @head:	the head for your list.
 * @member:	the name of the list_head within the struct.
 *
 * Continue to iterate over list of given type, continuing after
 * the current position.
 */
#define list_for_each_entry_continue(pos, head, member) 		\
	for (pos = list_next_entry(pos, member);			\
	     &pos->member != (head);					\
	     pos = list_next_entry(pos, member))

/**
 * list_for_each_entry_continue_reverse - iterate backwards from the given point
 * @pos:	the type * to use as a loop cursor.
 * @head:	the head for your list.
 * @member:	the name of the list_head within the struct.
 *
 * Start to iterate over list of given type backwards, continuing after
 * the current position.
 */
#define list_for_each_entry_continue_reverse(pos, head, member)     \
    for (pos = list_prev_entry(pos, member);            \
         &pos->member != (head);                     \
         pos = list_prev_entry(pos, member))

/**
 * list_for_each_entry_from - iterate over list of given type from the current point
 * @pos:    the type * to use as a loop cursor.
 * @head:   the head for your list.
 * @member: the name of the list_head within the struct.
 *
 * Iterate over list of given type, continuing from current position.
 */
#define list_for_each_entry_from(pos, head, member)            \
    for (; &pos->member != (head);                       \
         pos = list_next_entry(pos, member))

/**
 * list_for_each_entry_from_reverse - iterate backwards over list of given type
 *                                    from the current point
 * @pos:    the type * to use as a loop cursor.
 * @head:   the head for your list.
 * @member: the name of the list_head within the struct.
 *
 * Iterate backwards over list of given type, continuing from current position.
 */
#define list_for_each_entry_from_reverse(pos, head, member)     \
    for (; &pos->member != (head);                       \
         pos = list_prev_entry(pos, member))

/**
```

```
 * list_for_each_entry_safe - iterate over list of given type safe against removal of list entry
 * @pos:	the type * to use as a loop cursor.
 * @n:		another type * to use as temporary storage
 * @head:	the head for your list.
 * @member:	the name of the list_head within the struct.
 */
#define list_for_each_entry_safe(pos, n, head, member)			\
	for (pos = list_first_entry(head, typeof(*pos), member),	\
		n = list_next_entry(pos, member);			\
	     &pos->member != (head); 					\
	     pos = n, n = list_next_entry(n, member))

/**
 * list_for_each_entry_safe_continue - continue list iteration safe against removal
 * @pos:	the type * to use as a loop cursor.
 * @n:		another type * to use as temporary storage
 * @head:	the head for your list.
 * @member:	the name of the list_head within the struct.
 *
 * Iterate over list of given type, continuing after current point,
 * safe against removal of list entry.
 */
#define list_for_each_entry_safe_continue(pos, n, head, member) 		\
	for (pos = list_next_entry(pos, member), 				\
		n = list_next_entry(pos, member);				\
	     &pos->member != (head);						\
	     pos = n, n = list_next_entry(n, member))

/**
 * list_for_each_entry_safe_from - iterate over list from current point safe against removal
 * @pos:	the type * to use as a loop cursor.
 * @n:		another type * to use as temporary storage
 * @head:	the head for your list.
 * @member:	the name of the list_head within the struct.
 *
 * Iterate over list of given type from current point,
```

```c
safe against
 * removal of list entry.
 */
#define list_for_each_entry_safe_from(pos, n, head, member)             \
    for (n = list_next_entry(pos, member);                              \
         &pos->member != (head);                                        \
         pos = n, n = list_next_entry(n, member))

/**
 * list_for_each_entry_safe_reverse - iterate backwards over list safe against removal
 * @pos:    the type * to use as a loop cursor.
 * @n:      another type * to use as temporary storage
 * @head:   the head for your list.
 * @member: the name of the list_head within the struct.
 *
 * Iterate backwards over list of given type, safe against removal
 * of list entry.
 */
#define list_for_each_entry_safe_reverse(pos, n, head, member)      \
    for (pos = list_last_entry(head, typeof(*pos), member),      \
        n = list_prev_entry(pos, member);                   \
         &pos->member != (head);                            \
         pos = n, n = list_prev_entry(n, member))

/**
 * list_safe_reset_next - reset a stale list_for_each_entry_safe loop
 * @pos:    the loop cursor used in the list_for_each_entry_safe loop
 * @n:      temporary storage used in list_for_each_entry_safe
 * @member: the name of the list_head within the struct.
 *
 * list_safe_reset_next is not safe to use in general if the list may be
 * modified concurrently (eg. the lock is dropped in the loop body). An
 * exception to this is if the cursor element (pos) is
```

```
pinned in the list,
 * and list_safe_reset_next is called after re-taking the lock and before
 * completing the current iteration of the loop body.
 */
#define list_safe_reset_next(pos, n, member)			\
	n = list_next_entry(pos, member)

/*
 * Double linked lists with a single pointer list head.
 * Mostly useful for hash tables where the two pointer list head is
 * too wasteful.
 * You lose the ability to access the tail in O(1).
 */

#define HLIST_HEAD_INIT { .first = NULL }
#define HLIST_HEAD(name) struct hlist_head name = {  .first = NULL }
#define INIT_HLIST_HEAD(ptr) ((ptr)->first = NULL)
static inline void INIT_HLIST_NODE(struct hlist_node *h)
{
    h->next = NULL;
    h->pprev = NULL;
}

static inline int hlist_unhashed(const struct hlist_node *h)
{
    return !h->pprev;
}

static inline int hlist_empty(const struct hlist_head *h)
{
    return !READ_ONCE(h->first);
}

static inline void __hlist_del(struct hlist_node *n)
{
    struct hlist_node *next = n->next;
    struct hlist_node **pprev = n->pprev;

    WRITE_ONCE(*pprev, next);
    if (next)
```

```c
        next->pprev = pprev;
}

static inline void hlist_del(struct hlist_node *n)
{
    __hlist_del(n);
    n->next = LIST_POISON1;
    n->pprev = LIST_POISON2;
}

static inline void hlist_del_init(struct hlist_node *n)
{
    if (!hlist_unhashed(n)) {
        __hlist_del(n);
        INIT_HLIST_NODE(n);
    }
}

static inline void hlist_add_head(struct hlist_node *n, struct hlist_head *h)
{
    struct hlist_node *first = h->first;
    n->next = first;
    if (first)
        first->pprev = &n->next;
    WRITE_ONCE(h->first, n);
    n->pprev = &h->first;
}

/* next must be != NULL */
static inline void hlist_add_before(struct hlist_node *n,
                    struct hlist_node *next)
{
    n->pprev = next->pprev;
    n->next = next;
    next->pprev = &n->next;
    WRITE_ONCE(*(n->pprev), n);
}

static inline void hlist_add_behind(struct hlist_node *n,
                    struct hlist_node *prev)
{
    n->next = prev->next;
    WRITE_ONCE(prev->next, n);
    n->pprev = &prev->next;

    if (n->next)
        n->next->pprev  = &n->next;
}

/* after that we'll appear to be on some hlist and hlist_del will work */
static inline void hlist_add_fake(struct hlist_node *n)
{
    n->pprev = &n->next;
}

static inline bool hlist_fake(struct hlist_node *h)
{
    return h->pprev == &h->next;
}

/*
 * Check whether the node is the only node of the head without
 * accessing head:
 */
static inline bool
hlist_is_singular_node(struct hlist_node *n, struct hlist_head *h)
{
    return !n->next && n->pprev == &h->first;
}

/*
 * Move a list from one list head to another. Fixup the pprev
 * reference of the first entry if it exists.
 */
static inline void hlist_move_list(struct hlist_head *old,
                   struct hlist_head *new)
{
    new->first = old->first;
    if (new->first)
        new->first->pprev = &new->first;
    old->first = NULL;
}
```

```c

#define hlist_entry(ptr, type, member) container_of(ptr,type,member)

#define hlist_for_each(pos, head) \
    for (pos = (head)->first; pos ; pos = pos->next)

#define hlist_for_each_safe(pos, n, head) \
    for (pos = (head)->first; pos && ({ n = pos->next; 1; }); \
         pos = n)

#define hlist_entry_safe(ptr, type, member) \
    ({ typeof(ptr) ____ptr = (ptr); \
       ____ptr ? hlist_entry(____ptr, type, member) : NULL; \
    })

/**
 * hlist_for_each_entry - iterate over list of given type
 * @pos:    the type * to use as a loop cursor.
 * @head:   the head for your list.
 * @member: the name of the hlist_node within the struct.
 */
#define hlist_for_each_entry(pos, head, member)                \
    for (pos = hlist_entry_safe((head)->first, typeof(*(pos)), member);\
         pos;                           \
         pos = hlist_entry_safe((pos)->member.next, typeof(*(pos)), member))

/**
 * hlist_for_each_entry_continue - iterate over a hlist continuing after current point
 * @pos:    the type * to use as a loop cursor.
 * @member: the name of the hlist_node within the struct.
 */
#define hlist_for_each_entry_continue(pos, member)            \
    for (pos = hlist_entry_safe((pos)->member.next, typeof(*(pos)), member);\
         pos;                           \
         pos = hlist_entry_safe((pos)->member.next,
typeof(*(pos)), member))

/**
 * hlist_for_each_entry_from - iterate over a hlist continuing from current point
 * @pos:    the type * to use as a loop cursor.
 * @member: the name of the hlist_node within the struct.
 */
#define hlist_for_each_entry_from(pos, member)                  \
    for (; pos;                         \
         pos = hlist_entry_safe((pos)->member.next, typeof(*(pos)), member))

/**
 * hlist_for_each_entry_safe - iterate over list of given type safe against removal of list entry
 * @pos:    the type * to use as a loop cursor.
 * @n:      another &struct hlist_node to use as temporary storage
 * @head:   the head for your list.
 * @member: the name of the hlist_node within the struct.
 */
#define hlist_for_each_entry_safe(pos, n, head, member)         \
    for (pos = hlist_entry_safe((head)->first, typeof(*pos), member);\
         pos && ({ n = pos->member.next; 1; });         \
         pos = hlist_entry_safe(n, typeof(*pos), member))

#endif

```

```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);
    node = node->next;
  }
}

int GetSurvivor(const LinkedList *list) {
  assert(IsSingleton(list));

  return GetHeadVal(list);
}
```

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

~~~
# `13-linked-list`

- CMakeLists.txt (`ll/ll.c`)
- `struct node *`
- `#ifndef`

## Intrusive list

- [Data Structures in the Linux Kernel: Doubly linked list](https://0xax.gitbooks.io/linux-insides/content/DataStructures/linux-datastructures-1.html)
  - [types.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/types.h)
    - [list_head](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/types.h#L184)
  - [list.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/list.h)
  - [misc.c as an application](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/drivers/char/misc.c)

```c++
struct list_head {
  struct list_head *next, *prev;
};
```

## [linux](https://github.com/torvalds/linux)

- [types.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/types.h)
  - [list_head](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/types.h#L184)
- [list.h](https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/include/linux/list.h)
- [How does the kernel implements Linked Lists?](https://kernelnewbies.org/FAQ/LinkedLists)
  > Also illustrate how to use the list_head structure.
~~~

~~~markdown

## [`GList`](https://gitlab.gnome.org/GNOME/glib/-/blob/bc56578a087fc4eda0204b361d75162a4144546d/glib/glist.c) in

`GNOME/glibc`

- [glist.h](https://gitlab.gnome.org/GNOME/glib/-/blob/main/glib/glist.h)
- [glist.c](https://gitlab.gnome.org/GNOME/glib/-/blob/main/glib/glist.c)

```c++
typedef struct _GList GList;

struct _GList
{
  gpointer data;
  GList *next;
  GList *prev;
};
```

- [Docs for List](https://docs.gtk.org/glib/struct.List.html)

## `glibc`

- [list_t.h](https://github.com/bminor/glibc/blob/master/include/list_t.h)
- [list.h](https://github.com/bminor/glibc/blob/master/include/list.h)
~~~