C99-style initializers

```
struct hello_data {
        int tcount;
        struct list_head tlist;
};

static struct hello_data data = {
        .tlist = LIST_HEAD_INIT(data.tlist),
};

enum { STATUS_OK, STATUS_SO_SO, STATUS_BAD, STATUS_CODE_MAX};
const char *status_names[STATUS_CODE_MAX] = {
    [STATUS_OK]  = "All OK",
    [STATUS_BAD] = "All bad",
};

static struct hlist_head htable[HTABLE_SIZE] = {
    [ 0 ... HTABLE_SIZE - 1 ] = HLIST_HEAD_INIT,
};
```

Linked lists in Linux Kernel

- Most common, simple and convenient data structures
- In general developers are free in implementation choice. They can either use their own data structure and list manipulation primitives (iterators, insertion/deletion helpers, etc), but it is first better to look at Linux kernel “standard” linked list implementation if it suits task needs.
- Lists can also be double linked, where each node has pointer to previous one. Also list can terminate with NULL or point to the same stub head.
- In most simple case, singly directed linked list might look as following:

```
struct my_data {
        struct my_data *next;
        unsigned long canary;
};
```

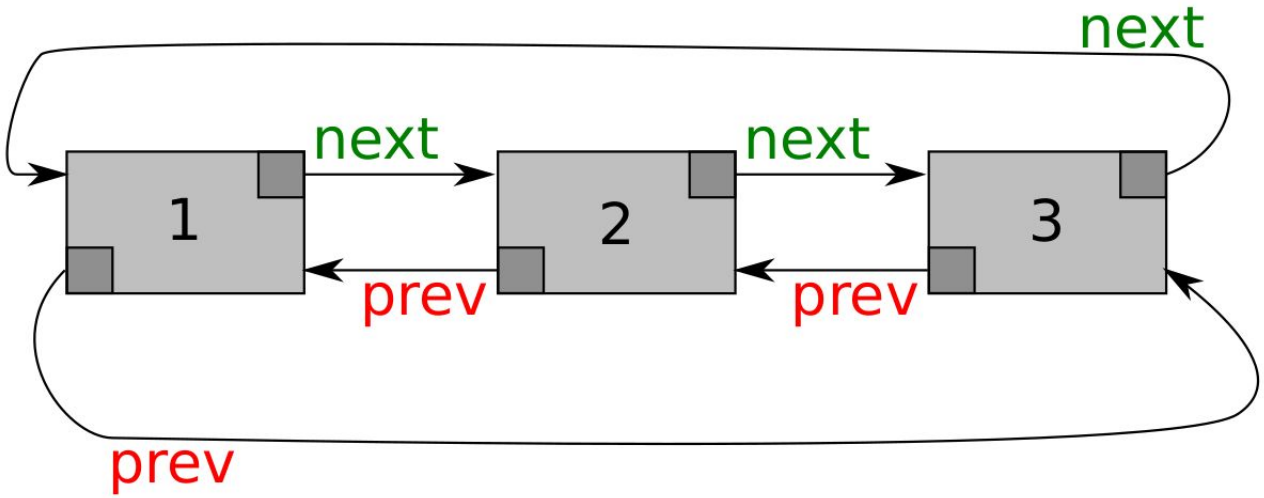
Linked lists in Linux Kernel (cont.)

- Standard Linux Kernel linked lists implemented as doubly linked circular lists.
- In common cases they use stub head to hold reference to the whole list, not just pointer to the first (last) element of the list. This property is used by most of list manipulation primitives as well as iterators.
- These lists are generic, embeddable into customers data structure turning these structures in a linked list.
- Linked list type is so common in kernel, so it is declared in `<linux/types.h>`.
- General linked list manipulation primitives are declared in `<linux/list.h>`.
- There is RCU variant list manipulation primitives in `<linux/rculist.h>`.
- As expected O(N) complexity is for list traversal and O(1) for list manipulation

Linked lists in Linux Kernel (cont.)

- struct list_head definition (see linux/types.h)

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



Linked lists in Linux Kernel (cont.)

- Here is brief overview on how to turn custom data structure into linked list that can be manipulated by standard Linux Kernel linked list primitives.

```
#include <linux/types.h>
#include <linux/list.h>

struct my_data {
    struct list_head list_node;
    struct list_head another_list_node;
    unsigned long canary;
};
```

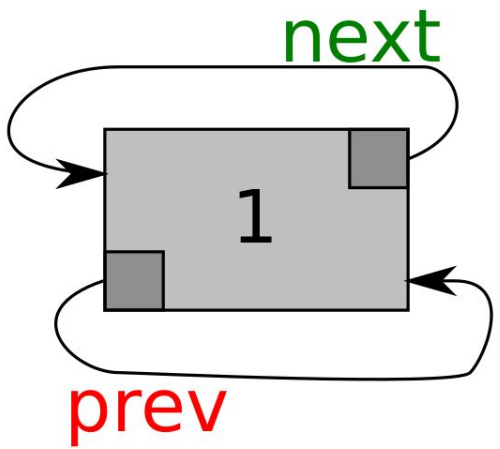
Linked lists in Linux Kernel (cont.)

- And here is how to define Linux Kernel linked list head

```
/* if list is global this can be used to define stub list head */
static LIST_HEAD(my_list_head);

/* which is in turn equivalent to */
static struct list_head my_list_head = LIST_HEAD_INIT(my_list_head);

static struct my_data *cool_init_function(void)
{
     /* here @os some other struct
      * containing stub list head is
      * allocated at runtime (e.g. in heap)*/
     INIT_LIST_HEAD(&os->my_list_head);
}
```



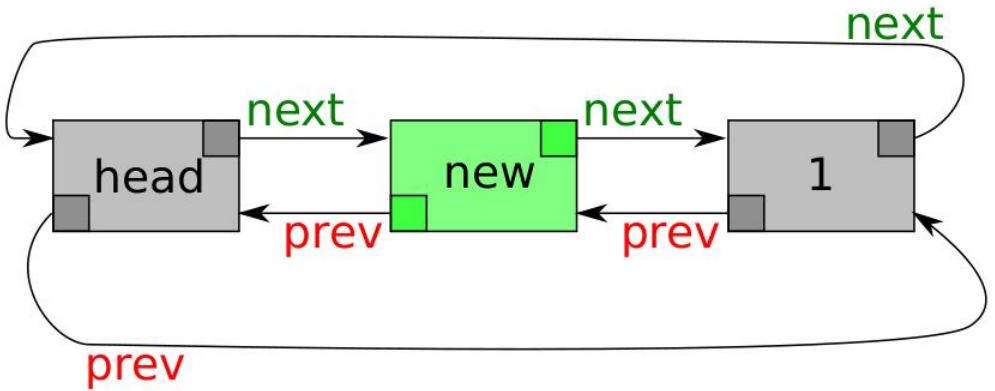
Linked lists in Linux Kernel (cont.)

- Let’s add some static data to the static, global stubby list head

```
struct LIST_HEAD(my_list_head);

/* static array of lists! */
static struct my_list ml[] = {
    { .canary = 0xfade0f01, },
    { .canary = 0xfade0f02, },
    { .canary = 0xfade0f03, },
};

/* somewhere in .text */
for (i = 0; i < ARRAY_SIZE(ml); i++) {
    list_add(&ml[i].list_node,
&my_list_head);
}
```



Linked lists in Linux Kernel (cont.)

- Now to test if list is empty one can use list_empty()

```
static LIST_HEAD(my_list_head);

static int init_my_data_list(unsigned int nr_items)
{
     /* Calling this function with non-empty
      * list head isn't supported for now.
      */
     BUG_ON(!list_empty(&my_list_head));
     /* rest of the code goes here */
}
```

- Internally list_empty(head) is simple inline helper in `<linux/list.h>` that returns @true if head->next == head and @false overwise.

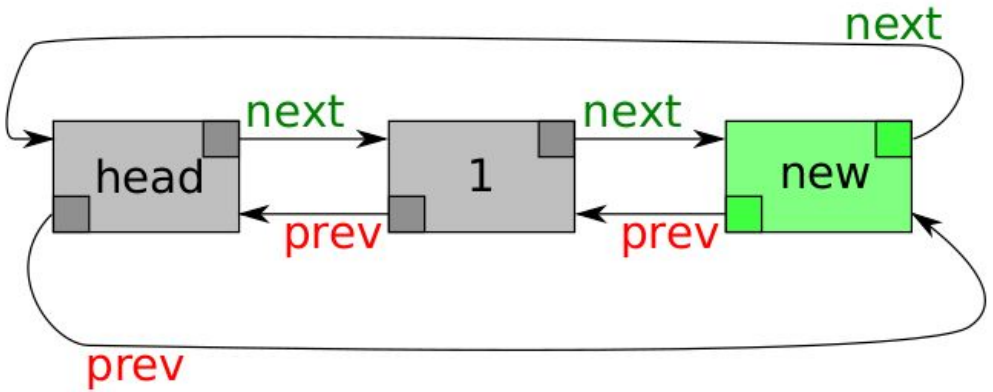
Linked lists in Linux Kernel (cont.)

- And to add in @canary increasing order using list_add_tail()

```
struct LIST_HEAD(my_list_head);

/* static array of lists! */
static struct my_data md[] = {
    { .canary = 0xfade0f01, },
    { .canary = 0xfade0f02, },
    { .canary = 0xfade0f03, },
};

/* somewhere in .text */
for (i = 0; i < ARRAY_SIZE(md); i++) {
    list_add_tail(&md[i].list_node, &my_list_head);
}
```



Linked lists in Linux Kernel (cont.)

- Okay, I want to replace item at known position within list with new one

```
struct my_data new_md = {
    .canary = 0xfade0ff,
};

list_replace(&md[1].list_node, &new_md.list_node);
```

- Note that there most of the routines can be called in empty list safely.
- To check if list is empty there is special helper list_empty() that compares head->next == head:

```
if (!list_empty(&my_list_head))
    /* do something that relies on non-empty list */
```

Linked lists in Linux Kernel (cont.)

- To splice together two lists one can use list_splice() or list_splice_tail()

```
static struct my_data *odd_canary(unsigned int nr_items);
static struct my_data *even_canary(unsigned int nr_items);

static LIST_HEAD(my_list_head);
static LIST_HEAD(my_list_even), LIST_HEAD(my_list_odd);

/* somewhere in the .text */
even_canary(&my_list_even, 10)
odd_canary(&my_list_odd, 10);

list_replace_init(&my_list_even, &my_list_head);
list_splice_tail_init(&my_list_odd, &my_list_head);
```

Linked lists in Linux Kernel (cont.)

- Now let’s delete something from the list using list_del()

```
list_del(&ml[1].list_node);

/* and then flush list completely */
struct my_data *md, *tmp;
list_for_each_entry_safe(md, tmp, my_list_head, list_node) {
        list_del(&md->list_entry);
}

/* check that list is really empty after flush via BUG_ON() */
BUG_ON(!list_empty(my_list_head));
```

- Note there is no need to reinitialize @my_list_head after delete since it becomes empty like after calling INIT_LIST_HEAD() on it.

Linked lists in Linux Kernel (cont.)

- Following routines are used to help to get parent (container) structure from the pointer to the list node and they are basis for rest of the primitives (mostly list iterators):

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
#define list_first_entry(ptr, type, member) \
    list_entry((ptr)->next, type, member)
#define list_last_entry(ptr, type, member) \
    list_entry((ptr)->prev, type, member)
#define list_next_entry(pos, member) \
    list_entry((pos)->member.next, typeof(*(pos)), member)
#define list_prev_entry(pos, member) \
    list_entry((pos)->member.prev, typeof(*(pos)), member)
#define list_first_entry_or_null(ptr, type, member) \
    (!list_empty(ptr) ? list_first_entry(ptr, type, member) : NULL)
```

Linked lists in Linux Kernel (cont.)

- Now let’s define list manipulation routines explicitly

```
/* Note that all of these functions very simple and thus inline functions in
 * <linux/list.h>. We just omit “static” and “inline” qualifiers here.
 */
int list_empty(const struct list_head *head);
void list_add(struct list_head *new, struct list_head *head);
void list_add_tail(struct list_head *new, struct list_head *head);
void list_del(struct list_head *entry);
void list_del_init(struct list_head *entry);
void list_replace(struct list_head *old, struct list_head *new);
void list_replace_init(struct list_head *old, struct list_head *new);
void list_move(struct list_head *list, struct list_head *head);
void list_move_tail(struct list_head *list, struct list_head *head);
void list_splice(const struct list_head *list, struct list_head *head);
void list_splice_tail(struct list_head *list, struct list_head *head);
void list_splice_init(struct list_head *list, struct list_head *head);
void list_splice_tail_init(struct list_head *list, struct list_head *head);
```

Linked lists in Linux Kernel (cont.)

- When talking about list iterators to travel linked list we can assume following
  - There are two classes
    - First presents list node parent data structure item in the body.
    - Second, less common presents list node itself in the list. Use list_entry() primitive to get pointer to the parent data structure when required, or (better) use one of the iterators from the first class.
  - There are support to travel forward/backward in the list
  - There are support to start from given entry or from next entry in the list. Useful to resume iterations in case of break
  - There are variants “safe” against list element deletion during iteration
  - After iterator completes reaching end-of-list, cursor pointer is **never** NULL
  - They just plain C for() loop statement

Linked lists in Linux Kernel (cont.)

- Here is list of iterators

```
/* First class. Presents parent structure pointed by @pos in iterator body.
 *
 * @pos - pointer to parent struct containing struct list_head @member.
 * @n - same type as @pos, but used to store next entry in the list.
 * @head - stub list head where iterations will stop.
 * @member - field of struct list_head type in structure pointed by @pos
 */
#define list_for_each_entry(pos, head, member)
#define list_for_each_entry_reverse(pos, head, member)
#define list_for_each_entry_continue(pos, head, member)
#define list_for_each_entry_continue_reverse(pos, head, member)
#define list_for_each_entry_from(pos, head, member)
#define list_for_each_entry_safe(pos, n, head, member)
#define list_for_each_entry_safe_continue(pos, n, head, member)
#define list_for_each_entry_safe_from(pos, n, head, member)
#define list_for_each_entry_safe_reverse(pos, n, head, member)
```

Linked lists in Linux Kernel (cont.)

- Here is list of iterators (cont.)

```
/* Second class. Presents list_head structure pointed by @pos in iterator body.
 *
 * @pos - pointer to parent struct containing struct list_head @member.
 * @n - same type as @pos, but used to store next entry in the list.
 * @head - stub list head where iterations will stop.
 */
#define list_for_each(pos, head)
#define list_for_each_prev(pos, head)
#define list_for_each_safe(pos, n, head)
#define list_for_each_prev_safe(pos, n, head)
```

Linked lists in Linux Kernel (cont.)

- The most common pitfall when searching for something and return pointer

```
static struct my_data *find_by_canary(unsigned int canary)
{
    struct my_data *md = NULL;

    list_for_each_entry(md, &my_list_head, list_node) {
        if (md->canary == canary)
            break;
    }

    return md;
}
```

Linked lists in Linux Kernel (cont.)

- The most common pitfall when using routine to populate list

```
static struct list_head *prepare_list_on_stack(unsigned int nr_items)
{
    LIST_HEAD(head);
    unsigned int i;
    for (i = 0; i < nr_items; i++) {
        struct md_data *md;
        /* alloc @md in heap */
        list_add(&md->list_node, &head);
    }
    /* it is illegal to return pointer to stack data, however
     * if you list_del(&head) here you need to list_add() new head
     * in the caller. */
    return &head;
}
```