EPITA - Practical Programming

C Programming - 04

Intrusive Structure

Generic Data Structure

Generic in C ?

- using void* equivalence
  - risky (lot of cast)
  - requires separate allocations
- using macro
  - almost unreadable
  - hard to write and maintain

Intrusive Structure ?

Put structures in data:

- One allocation per data
- Data cell can belongs to multiple containers
- Almost transparent implementations

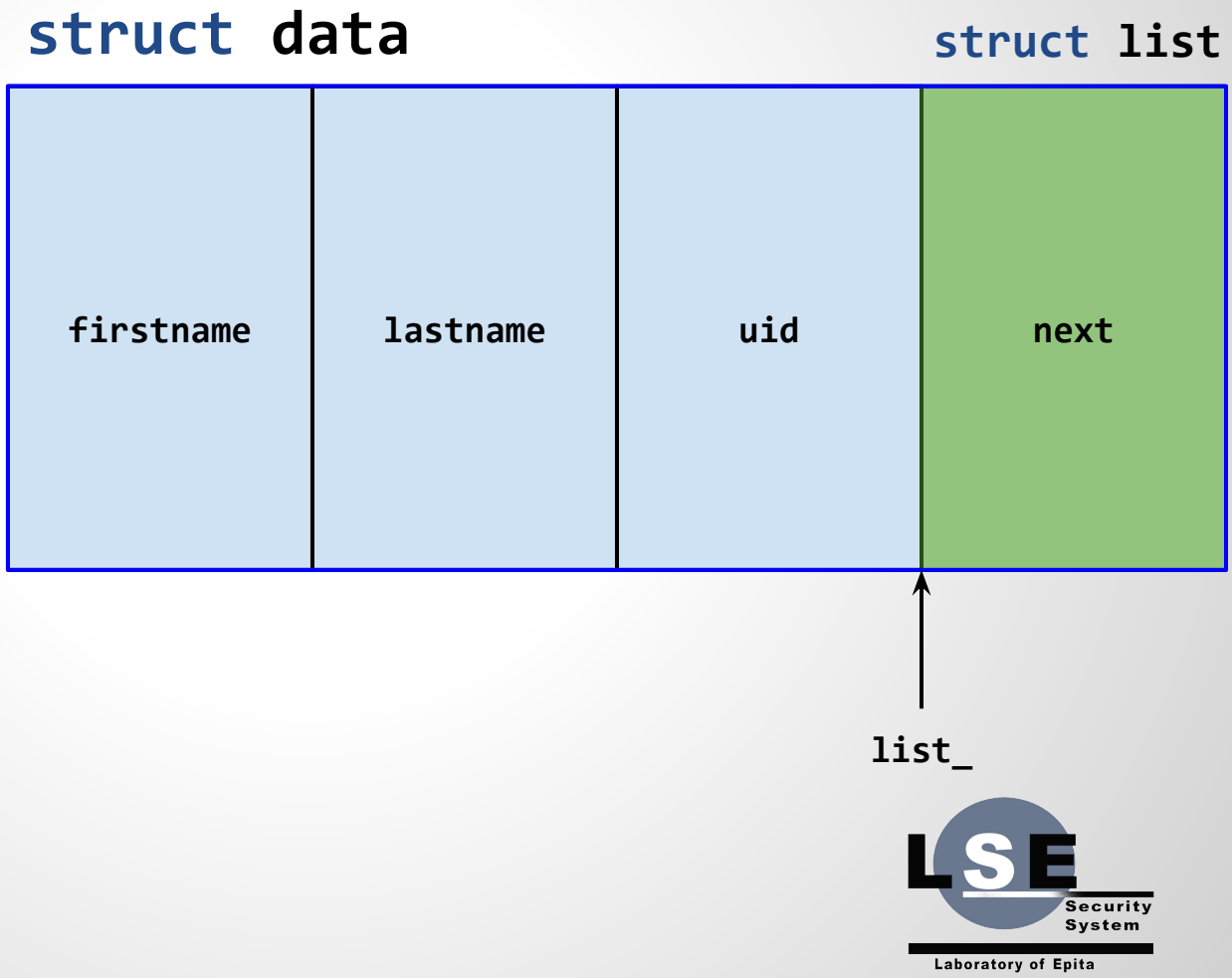
Real life example: list structures in Linux kernel

Structures Layout

Structures in Structures

```
struct list {
  struct list   *next;
};

struct data {
  char          *firstname;
  char          *lastname;
  unsigned       uid;
  struct list    list_;
};
```



LSE Security System
Laboratory of Epita

Access to parent pointer

- We have a pointer to the `_list` field
- We want the pointer to the outer struct
- We can use `offsetof` from `<stddef.h>`

```
#define CONTAINER_OF_(TYPENAME_, FIELDNAME_, PTR_)                    \
  ((TYPENAME_*)(((char*)PTR_ - offsetof(TYPENAME_, FIELDNAME_))))
```

LSE Security System
Laboratory of Epita

Intrusive Lists

Intrusive Lists

- Simple list implementation
- External allocation
- Sentinel (not mandatory but useful)
- Few tricks and almost no cast

Intrusive Lists

```c
struct list {
  struct list          *next;
};

// Now, compute the parent pointer
# define CONTAINER_OF_(TYPENAME_, FIELDNAME_, PTR_) \
  ((TYPENAME_*)(((char*)PTR_ - offsetof(TYPENAME_, FIELDNAME_))))

/*
 * list_init(sentinel)
 * Initialize list sentinel
 */
static inline
void list_init(struct list *sentinel) {
  sentinel->next = NULL;
}

/*
 * list_is_empty(l)
 * test for empty list (doesn't work for uninitialized list)
 */
static inline
int list_is_empty(struct list *l) {
  return l->next == NULL;
}
```

LSE Security System
Laboratory of Epita

Intrusive Lists

```c
/*
 * list_push_front(l, cell)
 * add cell in front of the l
 * (keep sentinel unchanged)
 */
static inline
void list_push_front(struct list *l, struct list *cell) {
  cell->next = l->next;
  l->next = cell;
}

/*
 * list_pop_front(l)
 * extract and return the first element of the list
 * returns NULL if the list is empty
 * (keep sentinel unchanged)
 */
static inline
struct list* list_pop_front(struct list *l) {
  struct list          *cell = l->next;
  if (cell) {
    l->next = cell->next;
    cell->next = NULL;
  }
  return cell;
}
```

LSE Security System
Laboratory of Epita

For Each

```
# define foreach(head_, cur) \
 for (struct list *cur = (head_); cur->next; cur = cur->next)

size_t len(struct list *head) {
  size_t                 l = 0;
  foreach(head, it) {
    l += 1;
  }
  return l;
}
```

For Each

```
struct cell {
  int           value;
  struct list   list_;
};

void print_list(struct list *head) {
  foreach(head, it) {
    printf("%d;", CONTAINER_OF_(struct cell, list_, it->next)->value);
  }
}
```

Advanced For Each

```
# define foreach(TYPENAME_, FIELD_, head_, elm_)                              \
  struct list *cur__;                                                         \
  for (cur__ = (head_), elm_ = CONTAINER_OF_(TYPENAME_, FIELD_, cur__->next); \
       cur__->next != NULL;                                                   \
       cur__ = cur__->next,                                                   \
       elm_ = CONTAINER_OF_(TYPENAME_, FIELD_, cur__->next))

void print_list(struct list *head) {
  struct cell  *elm;
  foreach(struct cell, list_, head, elm) {
    printf("%d;", elm->value);
  }
}
```

LSE Security System
Laboratory of Epita

More ?

LSE Security System
Laboratory of Epita

We Can Do More !

- You can integrate more than one container
- Intrusive approach can be used with any kind of linked containers (like trees)

LSE Security System
Laboratory of Epita

Other Approach

- Rather than computing offset, we can integrate the parent pointer in the list structure.
- Such implementation can be done using `void*` based lists.

Even simpler intrusive list

- Another solution is to simply include the list structure as first member
- CONTAINER_OF_ macro is now a simple cast
- With this technique data cell can only appears in one kind of container.

Memory Management

Reference Counting

- Memory management of structured shared by multiple container can become complex
- Life time of memory cell depends on references
- Simple Ref-counting can save a lot of issues.

Reference Counting

```
typedef void (*delete_handle)(void*);

struct refcount {
  size_t        count;
  delete_handle delete;
};

static inline
void refcount_init(struct refcount *ref,
delete_handle del) {
  ref->count = 1;
  ref->delete = del;
}
```

```
static inline
void refcount_decr(struct refcount *ref) {
  ref->count -= 1;
  if (ref->count == 0)
    ref->delete(ref);
}

static inline
void refcount_incr(struct refcount *ref) {
  ref->count += 1;
}
```

LSE Security System
Laboratory of Epita

Usage Example

```
struct data {
  int                   value;
  struct list           list_;
  struct refcount       rcount;
};

void delete_data(void *p) {
  free(CONTAINER_OF_(struct data, rcount, p));
}

struct data* create_cell(int x) {
  struct data           *cell;
  cell = malloc(sizeof (struct data));
  cell->value = x;
  refcount_init(&cell->rcount, delete_data);
  return cell;
}
```

Usage Example

```
struct list* build_int_list(int len) {
  struct list          *sentinel;
  sentinel = malloc(sizeof (struct list));
  list_init(sentinel);
  for (int i = 0; i < len; ++i) {
    struct data        *cell;
    cell = create_cell(i);
    // don't need to increment refcount, the only reference is inside the list
    list_push_front(sentinel, &cell->list_);
  }
  return sentinel;
}

void delete_list(struct list *head) {
  while (!list_is_empty(head)) {
    struct data        *cell;
    cell = CONTAINER_OF_(struct data, list_, list_pop_front(head));
    // Just decrement refcount
    // If no other ref is available, will free the pointer correctly
    refcount_decr(&cell->rcount);
  }
  free(head); // Sentinel is not ref-counted delete it explictely
}
```

LSE Security System
Laboratory of Epita